Electronic Notes in Theoretical Computer Science 82 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 15 pages

Coverage-driven Automated Compiler Test Suite Generation

A. Kalinov [1] A. Kossatchev [2] A. Petrenko [3] M. Posypkin [4]
V. Shishkov [5]

Institute for System Programming of Russian Academy of Sciences,
109004, Moscow, Russia, B. Kommunisticheskaya 25

Abstract

The paper presents a novel approach to automated compiler test suite generation based on the source level specification. Several coverage criteria are introduced. The application of the proposed methodology to testing the realistic programming language is discussed.

1 Introduction

Developing an adequate set of tests also called a *test suite* is an important part of the software development process. We faced this problem while working on the mpC parallel programming language [18] compiler. The general task was to develop a test suite for checking whether the particular compiler implementation correctly processes the source language.

In this case study we focus on testing language expressions. mpC provides powerful operators for array-based and parallel computations. That is why mpC expressions are complicated and difficult to implement and require thorough testing. However the proposed techniques are also applicable to other parts of the language.

Our approach is a sort of "specification-based testing" [20,24,2]. We use the Montages formalism [15] for modeling mpC expressions semantics. The formal semantics of the visual Montages formalism is based on Xasm [5,14], a programming language, based on Gurevich's Abstract State Machines [9] and Tree Finite State Machines, a concept recently introduced by Kutter [21]. The

[1] ka@ispras.ru
[2] kos@ispras.ru
[3] petrenko@ispras.ru
[4] posypkin@ispras.ru
[5] vova@ispras.ru

semantics has been implemented using the Gem-Mex tool, which is the part of the Xasm open source project [1].

We use the source language specification for three different purposes:

- **Test cases generation.** We use the static semantics part of the specification (syntax productions, constraints) to generate both a set of statically correct, and a set of statically incorrect programs, which constitute a positive and a negative test suite, respectively.
- **Test oracle generation.**The dynamic semantics part of the specification (execution behavior) is used for generating trustable output of a test program. Test oracle compares actual and trustable outputs for a particular test. If results are not identical the verdict is failure.
- **Providing test coverage criteria.** The specification coverage analysis demonstrates whether all parts of the specification are exercised by the test suite. If the coverage criteria are satisfied then no more test cases are needed, otherwise additional test programs should be added to the test suite.

The approach discussed in this paper is an improvement of our previous results [4,3]. Two new results are introduced:

- Several coverage notions for Montages specifications are introduced and discussed.
- An approach for generating negative test cases is presented.

The paper is organized as follows: Section 2 explains how mpC expression semantics was defined using Montages. Coverage criteria are introduced in Section 3. The architecture of the test suites is presented in Section 4. Section 5 overviews the test generation process. Practical results and related work are discussed in Sections 6 and 7 respectively. Section 8 overviews possible directions of future research.

2 The Montages Specification for mpC Expressions

2.1 Abstract State Machines

Abstract State Machine (ASM) is a new and powerful approach to specification of large-scale realistic software and hardware systems. We refer the reader to [9] for a detailed definition.

The state of an Abstract State Machine is given by the collection of functions on an abstract set called *superuniverse*. The basic ASM operation is an *update* which is defined as a function f value modification at a location given by a value of a tuple $< t_1, \ldots, t_r >$:

$$f(t_1, \ldots, t_r) := t_{r+1}$$

The ASM is driven by transition rules. The expression above called an *update rule* is a basic transition rule. More complex transition rules are obtained

by recursive application of *parallel composition* or *a conditional constructor*.

Parallel composition

The sequence of rules is a rule. The execution of a sequence of rules is defined as a simultaneous (parallel) execution of rules comprising the sequence (i.e. all updates defined by the rules take place simultaneously).

Conditional constructor

If $g_1, \ldots, g_k$ are Boolean terms and $R_1, \ldots, R_k$ are rules then the following expression is a rule:

$$\begin{array}{l} if\ g_1\ then\ R_1 \\ elseif\ g_2\ then\ R_2 \\ \vdots \\ elseif\ g_k\ then\ R_k \\ endif \end{array}$$

If at a given state S guard g_i holds and every g_j with $j < i$ fails then the execution of the rule described above is defined as the execution of the rule R_i.

2.2 Montages.

Montages [15] are a visual formalism for describing programming language syntax, static and dynamic semantics. Montages have been successfully used for the specification of Oberon [16], Java [25] and other programming languages. The formal semantics of the visual Montages formalism is based on Xasm [5,14], a programming language, based on Gurevich's Abstract State Machines [9] and Tree Finite State Machines, a concept recently introduced by Kutter [21].

The language specification is given as a collection of *Montages*. Each Montage defines the static and dynamic semantics of a particular language construct.

A Montage consists of several parts: a production rule, a local state machine, attribute definitions, constraint and dynamic semantics rules.

Static semantics is defined by the attribute grammar. After evaluating attributes the constraint is evaluated. If constraints for all AST nodes are evaluated to true the program is considered valid otherwise it is rejected.

The local state machine is expressed using MVL (*Montage Visual Language*). An MVL graph may contain nodes of three kinds:

- *Ovals.* Oval nodes represent states. The string associated with an oval is used to identify ASM actions which are triggered if the state is visited.

- *I and T.* Special nodes marked I and T denote entry and exit states of the machines.
- *Boxes.* Boxes represent the local state machines associated with siblings of a Montage. They are labeled in accordance with the corresponding right-hand-side non-terminals of the Montage's production rule.

Dotted arrows between nodes visualizes the *local control flow*, i.e. control flow within Montage. The local control flow defines an order in which states are visited.

Arrows may be labeled by boolean predicates called "firing conditions". The transition defined by an arrow is performed only if the corresponding predicate is evaluated to true.

For giving dynamic semantics, Montages use *Tree Finite State Machines (TFSM)* [21]. The idea of TFSM is to give a state machine where the states are tuples of a node in the AST, and a state in a simple machine, given here by MVL. For each node in the AST, only states of the MVL machine in the corresponding Montage are allowed. Transitions of TFSMs are quintuples:

(src_node, src_state, c, trg_node, trg_state)

leading control from the tuple *(src_node, src_state)* to the tuple *(trg_node, trg_state)* if condition *c* evaluates to true.

The local control flow arrows in the MVL machines can now be mapped into TFSM transitions where the source and target nodes, respectively states are determined by resolving the boxes and ovals being source and target of the MVL arrows.

The execution of a program according to its Montages semantics is defined as follows. Let the current state be denoted as *(CNode, CState)*.

- the execution starts in the TFSM state being built by the root of the AST, and the I-state of the MVL machine corresponding to that node. Thus *CNode* is the root, and *CState* is I.
- at every step of the execution:
  - the action associated with *CState* is executed, in an environment of the node *CNode*. Typically the action will read and write to attributes of *CNode*.
  - then a TFSM transition starting at *(CNode, CState)* is chosen, among those where the condition *c* evaluates to true. The condition evaluation may refer to both the source node *CNode*, and the target node of the transition.
  - the current node *CNode* is then updated to the target node of the chosen TFSM transition, and the current state *CState* is updated to the target state of the chosen TFSM transition.

Example 2.1 Consider a simple language defined by only two Montages (Figure 1). Figure 2 demonstrates language expression $1 + 2$, corresponding AST and the visual representation of the resulting TFSM execution.

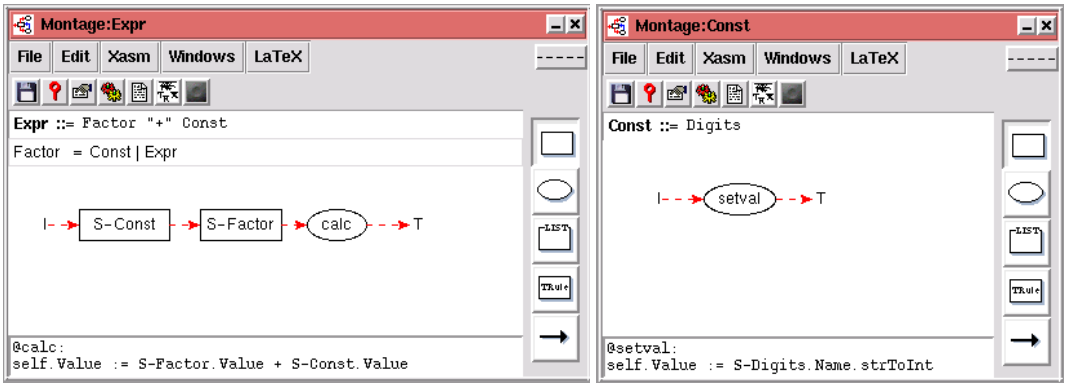

Fig. 1. Montages for a sample language.

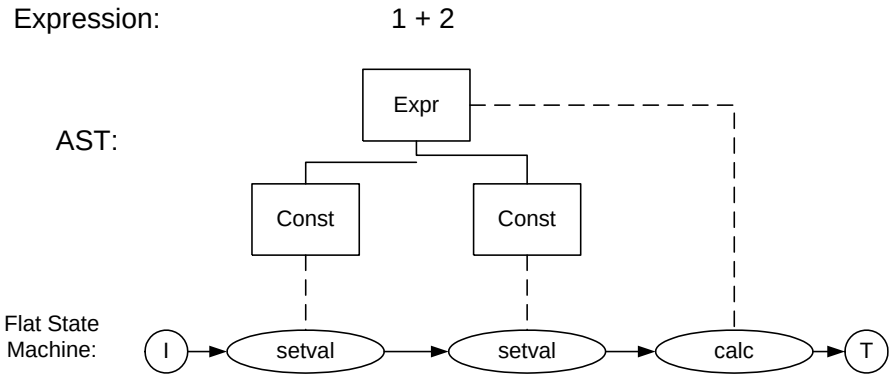

Fig. 2. TFSM execution for expression 1 + 2.

3 Montages-oriented Coverage Criteria

Below several coverage criteria for Montages specifications are introduced. In the sequel we assume that $\mathfrak{S}$ is a specification under test, $\mathfrak{L}$ is a language defined by Montages production rules and $\mathfrak{M} = \{M_1, \ldots, M_n\}$ is a set of all Montages in $\mathfrak{S}$. The set comprising local state machines states for all Montages from $\mathfrak{M}$ is denoted by $ST(\mathfrak{M})$.

We will often use the word "*fires*" instead of "executes" when talking about ASM rules or instead of "is visited" when talking about TFSM states.

3.1 ASM Coverage.

Coverage notions for ASM have been studied in [2]. We consider *strong parallel rule coverage* which is one of the most powerful coverage criterion for ASM specifications. Below we just recall the notion introduced in [2].

Let $\{r_i\}, 1 \leq i \leq m$ be a set of ASM rules.

Definition 3.1 The n-tuple T of ASM rules from $\{r_i\}$ is called *unfireable* if all the n rules from T can never simultaneously fire.

Definition 3.2 The test suite satisfies the *strong parallel rule coverage* for a transition rule $\{r_i\}$ if for every n, $1 \leq n \leq m$ every n-tuple of rules from $\{r_i\}$

is either unfireable or is simultaneously fired by execution of at least one test sequence from the test suite.

The notion introduced above could be easily extended for the case of Montages specification.

Definition 3.3 An ASM rule constructed from update rules by the parallel composition is called *basic block*.

Definition 3.4 The *strong parallel rule coverage for the set* $\mathfrak{M}$ *of Montages* is satisfied if for every state st from $ST(\mathfrak{M})$ the strong parallel rule coverage is satisfied for the set $BB(R(st))$ of all basic blocks of the rule $R(st)$ associated with the state st.

The fulfillment of the strong parallel rule coverage criteria guarantees that any possible combination of update rules is exercised by the test suite. However those criteria addresses only one aspect of the dynamic semantic execution – firing ASM rules. To address another aspect – the TFSM states traversal additional coverage notions should be introduced.

3.2 Combining TFSM and ASM Rule Coverage

Definition 3.5 Let $st_1, \ldots, st_n$ be states from $ST(\mathfrak{M})$ and let $T_1, \ldots, T_n$ be ASM rules tuples, where $T_i \subseteq BB(R(st_i))$ for all $i, 1 \leq i \leq n$. The n-tuple $\{st_1 < T_1 >, st_2 < T_2 >, \ldots, st_n < T_n >\}$ is called *n-state path*.

Definition 3.6 The n-state path $\{st_1 < T_1 >, st_2 < T_2 >, \ldots, st_n < T_n >\}$ is called *fireable* if there is such program whose dynamic semantics execution sequentially visits states $st_1, \ldots, st_n$ and fires tuple T_i of ASM rules at the state st_i for all $i, 1 \leq i \leq n$.

Definition 3.7 The test suite satisfies the *n-state path coverage* for the set $\mathfrak{M}$ of Montages if for every fireable n-path there is such test program from the test suite whose dynamic semantics execution sequentially visits states $st_1, \ldots, st_n$ and fires tuple T_i of ASM rules at the state st_i for all $i, 1 \leq i \leq n$.

Definition 3.8 The test suite satisfies the *strong n-state path coverage* for the set $\mathfrak{M}$ of Montages if for every m, $1 \leq m \leq n$ the test suite satisfies m-state path coverage for $\mathfrak{M}$.

The direct consequence of the above definition is that strong parallel rule coverage is the strong 1-state path coverage (which is equivalent to 1-state path coverage).

The coverage criterion A *is stronger than* the coverage criterion B if any test suite satisfying A satisfies B. The n-state path coverage is not necessary stronger than $(n-1)$-state path coverage but the strong n-state path coverage is greater then strong $(n-1)$-state path coverage.

3.3 Coverage Notions for Semantics Constraints

The coverage criterion discussed in this section is aimed at measuring a quality of test suites constructed from semantically-incorrect tests.

The program is rejected if at least one of semantic constraints is violated. In most cases semantic constraint can be formulated as a simultaneous fulfillment of several conditions $C_1, \ldots, C_n$, called *basic conditions for a given constraint*. In this case the corresponding boolean expression looks as follows: $C_1 \& \ldots \& C_n$.

Thus constraint violation consists in violating of at least one of its basic conditions. The violation of each basic condition is a semantic error which should be correctly processed by the compiler. The fulfillment of the coverage criterion (definition 3.10) introduced below guarantees that all such errors are exercised by the test suite.

Definition 3.9 The *constraint coverage for a constraint* $C_1 \& \ldots \& C_n$ is satisfied if for any k, $1 \leq k \leq n$, there is a program from the test suite such as $C_k = false$ and $C_i = true$ for all i, $1 \leq i \leq n, i \neq k$.

Definition 3.10 The *constraint coverage for a set of Montages* $\mathfrak{M}$ is satisfied if for every Montage from $\mathfrak{M}$ the constraint coverage for its constraint is satisfied.

The following simple fact from first-order logic justifies that the fulfillment of the criterion 3.10 is always possible if every specification constraint does not contain redundant basic conditions.

Fact 3.11 *Let* $C_1 \& \ldots \& C_n$ *be a semantic constraint for some Montage from* $\mathfrak{M}$. *One of the following statements is true:*

(i) *There is a program from* $\mathfrak{L}$ *such that* $C_n = false$ *and* $C_1 = \cdots = C_{n-1} = true$ *for this program.*

(ii) $C_1 \& \ldots \& C_{n-1} \& C_n = C_1 \& \ldots \& C_{n-1}$ *for any program from* $\mathfrak{L}$ *(i.e. codition* C_n *is redundant)*

Informally it means that for well-designed specification the criterion 3.10 is always achievable.

If a generated test suite does not satisfy the criterion 3.10, i.e. for some constraint C the criterion 3.9 is not satisfied, then this constraint should be inspected and freed from redundant conditions. If no redundant conditions are detected then the test suite should be extended to meet the criterion 3.9 for constraint C.

Like this besides testing the implementation, negative tests help to identify the redundant conditions in the specification.

4 The Test Suite Architecture

Usually a compiler test suite consists of two sets of test cases: *positive test cases* and *negative test cases*. Further in the paper those sets are referred as *positive test suite* and *negative test suite* respectively.

Positive test cases are semantically and syntactically correct programs accompanied with their trustable outputs. A test program contains several initializations of variables being involved in the expression under test, expression itself and the "printf" function call for outputting the expression value. [6]

The positive test suite run is organized as follows. First, every program from the test suite is compiled by the compiler under test. Second, the obtained binary file is executed to produce *actual output*. Third, actual output is compared with trustable one. If outputs are not identical the verdict is failure.

Since we address only testing the implementation of language semantics not syntax the negative test cases are syntactically correct programs with one or more semantic mistakes.

The run of negative test suite is organized as follows. Each test program is compiled by the compiler under test. The test oracle checks whether the compiler recognizes the semantic error properly, i.e. the compiler reported an error rather then produced an output or crashed.

5 Generating the Test Suite from the Specification

The proposed scheme of test suite generation is depicted in Figure 3. We omit some technical details in order to make explanation clear and concise.

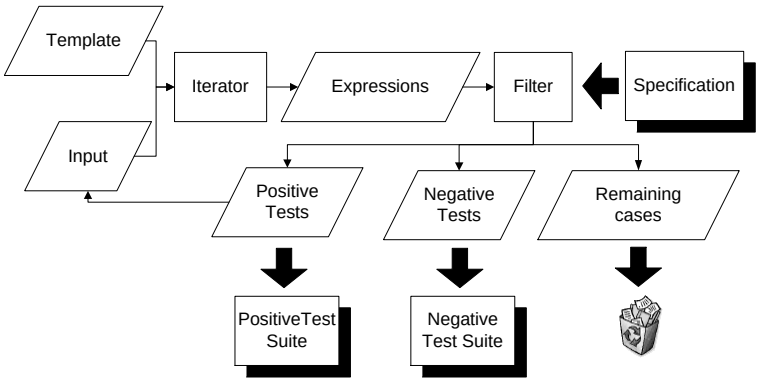

Fig. 3. The scheme of the test suite generation.

[6] Due to the space limitations we omit some technical details of constructing an oracle for a positive test case which could be found in [4]

5.1 Iterator

The *Iterator* produces syntactically correct mpC expressions from the *Template* and *Input* files. Then produced expressions are processed by the Montages specification and expressions violating semantics constraints are filtered out.

The *Iterator* generates test programs from two files: *Template* and *Input*. *Template* is a set of several mpC operators. *Input* is a set of several mpC expressions. The generation is implemented as a substitution of operands from *Input* as operands of operators from *Template*.

The initial contents of the *Input* file is provided by the *tester* (a person who organizes the test suite generation). It typically consists of basic expressions like constants and identifiers. Therefore first step of the generation produces only expressions containing one operator. At the second step expressions generated at the first step recognized as positive test cases are used as an *Input*. It makes possible the generation of expressions containing two operators. The third step uses expressions generated at the second step and so on.

5.2 Filter

The *Filter* carries out two tasks: separating correct and incorrect programs and filtering out test cases which do not increase coverage (see Figure 4).

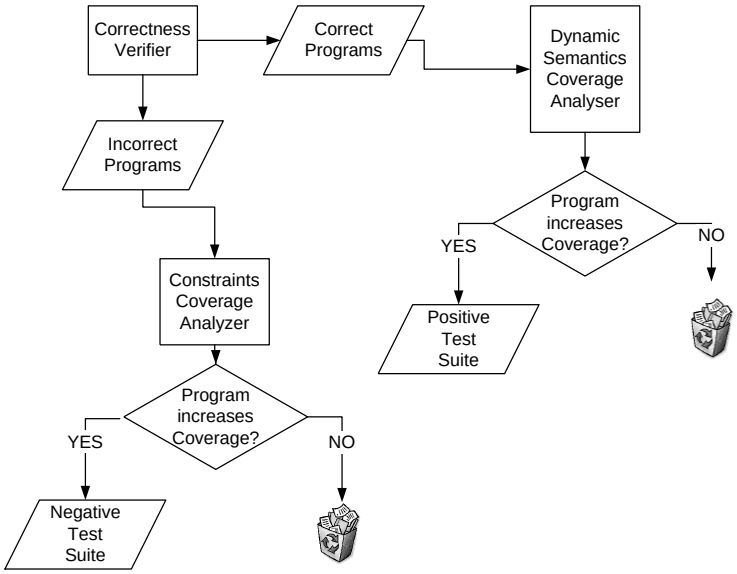

Fig. 4. The Filter's scheme.

The *Correctness Verifier* is the specification executable returning non-zero return code on incorrect test programs.

The *Constraint Coverage Analyzer* is based on the constraint coverage. For any constraint C, $C = C_1 \& \ldots \& C_n$ it constructs n-tuple of values of basic conditions C_i on a processing test program. If the tuple has exactly

one $false$ element and it does not belong to the set $T(C)$ of already obtained tuples for C then the tuple is added to the set C and the processing test program is appended to the negative test suite. Otherwise the program is filtered out and the set $T(C)$ is not modified.

The *Dynamic Semantics Coverage Analyzer* is based on strong n-state path coverage. If for some $1 \leq m \leq n$ the processing test program traverses m-paths which are absent in the set P of processed paths then all new traversed m-paths are added to P and the test is added to the positive test suite.

We can manage the generation process by tuning the coverage criterion. The stronger coverage criterion provides less restrictive filtering. One of the possibble options is to skip coverage checking at all.

6 Practical Results

The proposed approach was successfully applied to testing of the mpC parallel programming language compiler. More then 30 mistakes in both static and dynamic semantics processing were found and fixed. As a result the compiler quality was significantly improved.

Another practical outcome of proposed methodology was the improvement of the language documentation: some contradictions in the expression part of the language semantics were discovered. Along with testing the implementation the formal specification was also tested. We found several errors in the initial version of the mpC specification.

6.1 Positive Test Suites

mpC is a two-level ANSI C extension. The first level (also called C[]) [8] supports array-based computations in a spirit of FORTRAN 90 [19]. The language introduces special operators for manipulating arrays as a whole.

The second level extends C[] with facilities for parallel computations. mpC expressions could be used for expressing both computations and data exchange between different computing nodes. More information could be found in [18].

Our first attempt was to address only the C[] part. The idea was to start the generation with *Template* file consisting of all possible C[] operators and the *Input* file consisting of operands of all principally different types (all scalar types and vectors of one and two dimensions). The coverage tracking was not applied and negative tests were not addressed.

The first step of the test suite generation produced 135 test cases. New *Template* file was obtained by merging the initial contents of *Template* file and the set of generated expressions. The second run of the test generator produced 13473 test cases.

The analysis of failed tests demonstrated that step 2 introduces a new kind of errors (run-time error) and new errors of existing kinds. This confirms the intuitive idea that a test suite consisting of expressions with only one operator

is not sufficient for comprehensive testing.

The straightforward application of the same approach to testing the complete set of mpC operators faced significant troubles due to the dramatic increasing of the number of test cases. The first iteration produced 447 test cases thus making the second iteration impossible. The analysis of generated test suites showed that a lot of tests in fact test the same language feature and the amount of tests can be significantly decreased. This clearly indicated the necessity of applying the coverage criteria.

As in the case of C[] we started with the *Template* file consisting of all mpC operators and the *Input* file consisting of the set of mpC operands with all possible principally different combinations of types and distributions. Since the first iteration generates expressions containing one operator and the second iteration contain two or more operators we applied strong parallel rule coverage for the first iteration. First iteration produced 43 positive and 68 negative test cases. The obtained positive and negative test suites satisfied the strong parallel rule coverage and constraint coverage criteria respectively.

Applying strong parallel rule coverage decreases the number of generated test cases in 10 times (from 447 to 43). We compared compiler bugs detected by the both test suites. Surprisingly there was only one code-generator error in the large test suite (447 test cases) which was not detected by the small one while the total amount of distinct errors detected by the small test suite was more then 10. This result confirms that the strong parallel rule coverage criteria is a practically valuable coverage measure for the expressions containing one operator.

The situation with static semantics errors is less optimistic: the large test suite detected 4 new errors in the static semantics analyzer while the small one detected only 1 such error. This is an expectable result since the proposed coverage criteria are based on the dynamic semantics. Therefore testing of the static semantics processing requires the different coverage criteria. Maybe one of the approaches proposed in [10,17] should be applied.

The second iteration was driven by 2-state path coverage. There were generated 646 positive test cases. The number of negative test cases remained the same since the constrained coverage criterion was satisfied by the first iteration. The run of the generated test suites discovers a lot of different bugs in the compiler. The distribution of failed tests is summarized in the following table:

	1st Step	2nd Step
No Errors	19	180
Static Semantics	9	95
Code Generation	10	234
Segmentation Fault	1	45
Result Mismatch	0	13
Run-time Error	4	79

The positive test suite generated at the second iteration highlighted several bugs which do not appear on the first-level test suite. It justifies the necessity of introducing of n-state path coverage criteria.

6.2 Negative test suite

The negative test suite of the 68 test cases discovered 4 bugs in static semantics processing. The careful analysis of the coverage tracking results helps to detect several redundant conditions in the specification constraints.

7 Related Work

The specification-based testing is a well-known testing methodology addressed in [20], [11], [12], [24], [2]. The main benefit of the specification-based testing is the possibility of producing implementation-independent test suites. Obtained test suites are used for *conformance testing*.

Papers [24,2] address ASM-based testing: different coverage notions as well as strategies for automatic test suite generation are investigated. However the proposed approaches are aimed at testing of API or FSM-based systems.

There are several works addressing the compiler testing. Grammar-based approaches to the testing of the implementation of the language syntax and static semantics are addressed in [10,17,13]. Authors propose coverage criteria for attribute grammars and coverage oriented test generation strategies. Paper [10] considers negative test cases. The proposed approaches are suitable for testing the language static semantics, the dynamic semantics is not considered at all.

In the work [22] the approach based on the grammar transformations is presented. Authors successfully applied it for testing the compiler optimization transformations. The language grammar is transformed in order to address only language constructs which are in the scope of testing. This makes the automatic test generation feasible. However neither the coverage criteria nor the test oracle construction are addressed.

Several authors address the problem of generating oracles for compiler

tests [6,7,23]. The proposed approaches are aimed at checking preserving the original semantics of the transformed programs. The construction of the test oracle for positive test cases generated by the tool described in this paper is addressed in [4] in detail.

8 Future Work

The coverage-based generation proposed in this paper is a step towards reducing the number of redundant test cases and the time for the test suite generation. However the problem of a huge amount of test cases still exists.

For a typical language a very small percentage of syntactically correct programs are also semantically correct. Thus we can significantly reduce the time of tests generating by providing more "intelligent" *Iterator* producing less amount of semantically incorrect tests. Perhaps the *Iterator* should rely on syntactic as well as semantics structure of the language.

Unfortunately the fulfillment of n-state path coverage criteria for n greater then 1 is difficult to check since it is difficult to verify which combinations of states and firing tuples are possible. We also plan to address this problem in our future research.

Practical experiments demonstrate that the negative test suite discovered a relatively small amount of compiler errors. The preliminary experiments showed that not only the violating constraint but the semantics context of an error should be considered. Thus more powerful coverage notions for negative test cases should be developed.

9 Acknowlegments

We would like to thank Philipp Kutter for allowing us to use unpublished material from his dissertation and for providing us with examples and support using the Gem-Mex tool.

References

[1] eXtensible Abstract State Machines. www.xasm.org.

[2] A. Gargantini and E. Riccobene. ASM-based Testing: coverage criteria and automatic tests generation. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 262–265, February 2001.

[3] A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shishkov. Using ASM Specifications for Compiler Testing. In *Abstract State Machines - Advances in Theory and Applications 10th International Workshop, ASM 2003*, volume 2589 of *LNCS*, 2003.

[4] A. Kalinov, A. Kossatchev, M. Posypkin, V. Shishkov. Using ASM specification for automatic test suite generation for mpC parallel programming language compiler. In Peter D. Mosses, editor, *Proceedings of the Fourth International Workshop on Action Semantics, AS 2002,* (Copenhagen, Denmark, July 21, 2002), number NS-00-8 in Notes Series, pages 96–106, Department of Computer Science, University of Aarhus, December 2002. BRICS. vi+130 pp.

[5] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000.

[6] C. Jaramilio, R. Gupta, M.L. Soffa. Comparison checking: An approach to avoid debugging of optimized code. In *Proc. of ASM SIGSOFT*, volume 1687 of *LNCS*, pages 268–284, 1999.

[7] G. Necula. Translation validation for an optimizing compiler. In *Proc. of ASM SIGPLAN*, pages 83–95, Vancouver, Canada, 2000.

[8] Sergey Gaissaryan and Alexey Lastovetsky. ANSI C superset for vector and superscalar computers and its retargetable compiler. *Journal of C Language Translation*, 5, 1994.

[9] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[10] J. Harm and R. Lämmel. Two-dimensional approximation coverage. *Informatica*, 24(3), 2000.

[11] I. Bourdonov, A. Demakov, A. Jarov, A. Kossatchev, V. Kuliamin, A. Petrenko, S. Zelenov. Java specification extensions for automated test development. In *Proceedings of "Perspectives of System Informatics"*, volume 2244 of *LNCS*, pages 301–307, 2001.

[12] I. Bourdonov, A. Kossatchev, A. Petrenko, D. Galter. Automated generation of test suites from formal specifications. In *Proceedings of "World Congress of Formal Methods"*, volume 1708 of *LNCS*, pages 608–621, 1999.

[13] J.Hannan, E. Pfenning. Compiler verification in lf. In *Proceedings of 7th Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, 1992.

[14] P. Kutter. The Formal Definition of Anlauff's eXtensible Abstract State Machines. TIK-Report 136, Swiss Federal Institute of Technology (ETH) Zurich, June 2002.

A formal denotational semantics for XASM [5].

[15] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[16] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[17] Ralf Lämmel. Grammar testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[18] Alexey Lastovetsky, Dmitrij Arapov, Alexey Kalinov, and Ilya Ledovskih. A parallel language and its programming system for heterogeneous networks. *Concurrency: Practice and Experience*, 12:1317 – 1343, 2000.

[19] M. Metcalf and J. Ried. *Fortran 90 Explained*. Oxford University Press, 1992.

[20] A. Petrenko. Specification based testing: Towards practice. In *Proceedings of Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 287–300, 2001.

[21] P.W. Kutter. *Montages - Engineering of Computer Lnaguages*. PhD thesis, ETH Zurich, 2002.

[22] S. Zelenov, S. Zelenova, S. Kossatchev. Test generation for compilers and other text processors. *Programming and Computer Software*, (1), 2003.

[23] T.C. McNerney. Verifying the correctness of compiler transformations on basic blocks using abstract interpretation. In *Proc. of Simposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 106–115, 1991.

[24] W. Grieskamp and Y. Gurevich and W. Schulte and M. Veanes. Testing with Abstract State Machines. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 257–261, February 2001.

[25] C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

A specification of the static and dynamic semantics of Java, using ASMs and Montages [15].